CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 5

MENTOR:
L'ENVIRONNEMENT PASCAL

Bertrand MÉLÈSE

Octobre 1981

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: 954 90 20

MENTOR : L'ENVIRONNEMENT PASCAL

Bertrand MELESE
INRIA

RESUME

Ce rapport présente l'environnement de programmation Pascal qui existe sous MENTOR. Cet environnement est un ensemble de commandes de haut niveau dont le rôle est d'assister le programmeur durant toutes les phases de son travail : développement, mise au point, maintenance et transport des programmes Pascal. Quelques commandes, que nous pensons être représentatives de l'ensemble du système, sont décrites en détails. Pour les autres, un mode d'emploi est donné en appendice.

ABSTRACT

This report presents the Pascal programming environment based on the MENTOR system. This environment is a set of high level tools intended to assist the programmer at each step of his work : program development, debugging, transport and maintenance. Some of the most representative tools are fully described. A brief explanation of several other tools is given in an appendix.

MENTOR : L'ENVIRONNEMENT PASCAL

INTRODUCTION

Le système MENTOR ([DON75][HUE77][DON79][DON80][MUL81]) est un éditeur spécialisé dans la manipulation de données structurées. Il crée et utilise une représentation des données sous forme d'arbres étiquettés : les arbres de syntaxe abstraite. Actuellement, les données manipulées par MENTOR sont essentiellement des fragments de programmes Pascal. ([JEN74]).

Les travaux récents effectués sur MENTOR ont consisté en la définition et l'implémentation d'un ensemble cohérent de commandes de haut niveau pour la manipulation interactive, rapide, et sûre des programmes Pascal. ([MEL80]).

Certaines commandes agissent localement sur le programme et correspondent à l'automatisation des opérations les plus courantes en programmation :

- les déplacements dans le programme,
- les modifications des déclarations,
- l'introduction et la suppression d'instructions de traces,
- la création de commentaires ponctuant les différentes parties logiques du programme,
- la création et l'utilisation d'informations sur l'organisation générale du programme,
- etc...

D'autres commandes effectuent des transformations sur l'ensemble du programme. Leur utilisation est plus coûteuse et donc moins systématique. Elles tendent à banaliser des transformations qu'il serait long et dangereux de faire "à la main" :

- Transformations visant à normaliser la présentation des programmes, particulièrement utiles lorsqu'un logiciel est développé par une équipe de programmeurs.

- Alpha-conversion des programmes (décrite plus loin).
- Recensement, et suppression éventuelle, des objets déclarés dans un programme et qui ne sont pas utilisés.
- etc... .

La plupart des commandes disponibles sont listées en appendice. Ce sont toutes des procédures écrites en MENTOL qui est le langage de commande du système MENTOR.

L'utilisateur de MENTOR peut donc programmer en MENTOL d'autres outils, adaptés à ses problèmes particuliers, en utilisant toutes les commandes disponibles comme des primitives du langage MENTOL.

Le but de cet article est de faire sentir au lecteur les idées que nous pensons être importantes en matière d'aide à la programmation.

Pour cela, nous allons décrire quelques commandes représentatives de l'ensemble du système d'aide à la programmation proposé autour de MENTOR.

Dans la première partie nous décrivons très en détails une des commandes fréquemment utilisée : La commande DECVAR de déclaration de variables.

Dans la deuxième partie nous présentons quelques outils qui opèrent sur l'ensemble du programme.

Le système d'aide à la programmation complet est décrit en détails dans ([MEL80]).

Des procédures pour le calcul des effets de bords des procédures et des fonctions dans les programmes Pascal ont également été réalisées avec MENTOR ([MOR79]).

I - UN EXEMPLE DETAILLE : LA COMMANDE DECVAR

La commande DECVAR demande à l'utilisateur

- une liste de noms de variables (liste d'identificateurs),
- une expression de type.

Elle crée la portion de code Pascal (c'est-à-dire le sous-arbre) qui correspond à la déclaration de ces variables. Cette déclaration est ensuite insérée dans le programme. Nous allons voir dans la suite à quel endroit exact la nouvelle déclaration est insérée et quelles sont les vérifications faites par DECVAR.

I.1 - Notations

Nous appelons bloc une procédure, une fonction ou un programme Pascal, c'est-à-dire, sous MENTOR, tous les sous-arbres dont l'opérateur de tête est un opérateur BLOCK.

Nous appelons bloc courant, le bloc le plus interne qui contient le sous-arbre pointé par le pointeur courant de MENTOR.

Le pointeur courant est une variable de MENTOR qui indique toujours le sous-arbre sur lequel on est positionné à un instant donné. Nous le noterons : K. Nous noterons K l'occurrence pointée par le pointeur courant : K. D'une façon générale, nous noterons P l'occurrence pointée par une variable MENTOR : P.

Soit I une instruction du langage MENTOL. Soit : P une variable MENTOR désignant un sous-arbre du programme. Nous noterons I(P) l'occurrence pointée par : P après l'exécution de l'instruction I. Si l'instruction I ne modifie pas la variable MENTOR : P, on a I(P)=P.

Nous noterons B(P) le bloc le plus interne contenant la position P. Le bloc courant est donc le bloc B(K).

I.2 - Positionnement des déclarations

Dans DECVAR, l'option par défaut est d'insérer les nouvelles déclarations dans le bloc courant.

Le positionnement des déclarations peut être contrôlé par l'utilisateur en passant un argument à DECVAR au moment de l'appel. Nous appelerons cet argument "Niveau de Déclaration" ou ND. Lorsqu'il apparaît, l'argument ND est une instruction MENTOL qui représente un chemin dans l'arbre à partir de la position courante K. La déclaration est alors créée dans le bloc B(ND(K)), c'est-à-dire dans le bloc le plus interne contenant la position ND(K).

L'argument ND passé à DECVAR peut donc être vu comme un déplacement de la position courante avant l'exécution de DECVAR, la position courante étant ramenée à sa place initiale une fois l'exécution terminée.

Remarque :

La commande DECVAR crée une zone de déclaration de variables si cela est nécessaire, c'est-à-dire dans le cas ou aucune variable n'était préalablement déclarée dans le bloc concerné par les nouvelles déclarations.

I.3 - Vérification des déclarations

La commande DECVAR vérifie que les nouvelles variables déclarées n'introduisent pas de conflits avec des objets déjà existants dans le programme. Par "objets" nous entendons les constantes, types, variables, procédures ou fonctions.

I.3.1 - Définition d'un conflit

Nous distinguerons deux sortes de conflits : les conflits syntaxiques et les conflits sémantiques.

Conflits syntaxiques

Soit Q un bloc et v une variable à déclarer dans Q. Nous dirons que v crée un conflit syntaxique lorsqu'il existe un autre objet de Q qui porte le même nom que v. L'introduction de la déclaration de v dans Q crée une erreur qui sera détectée par le compilateur Pascal.

Conflits sémantiques

Nous dirons que la déclaration de v dans Q crée un conflit sémantique lorsqu'aucun objet v n'était préalablement déclaré dans Q mais que la nouvelle déclaration va "capter" des occurrences d'un objet v global à Q. Un tel conflit ne sera détecté par un compilateur Pascal que si il provoque une incohérence dans les types, c'est-à-dire, si l'objet v global n'est pas une variable ou n'est pas de même type que la variable v nouvellement déclarée.

I.3.2 - Les vérifications faites par DECVAR

I.3.2.1 - L'option de vérification par défaut

La commande DECVAR s'assure qu'aucun conflit, syntaxique ou sémantique, n'est introduit par la déclaration à créer.

Soit v l'une des variables à déclarer. La condition suivante est testée par DECVAR.

(C) "Aucun objet visible de l'intérieur du bloc B(ND(K)) n'a le même nom que v".

Si la condition (C) est vérifiée, la déclaration de v est acceptée. Sinon DECVAR demande un autre nom pour v.

Remarques sur la condition (C)

R1 : La condition (C) est suffisante pour assurer que v n'introduit aucun conflit.

R2 : La condition (C) n'est pas nécessaire. En effet, cette condition amènera parfois DECVAR à refuser des noms de variables qui n'auraient créé aucun conflit. Ce sera le cas chaque fois qu'une variable à déclarer aura le même nom qu'un objet global mais que cet objet n'a pas d'occurrence dans B(ND(K)).
Nous verrons plus loin que l'on peut remédier à cette situation en passant un argument de contrôle à DECVAR.

R3 : En revanche (C) a l'avantage d'unifier les vérifications faites pour détecter les conflits syntaxiques et sémantiques.

R4 : L'avantage le plus important, dans le cadre d'un système interactif, est que cette condition peut être testée assez rapidement tout en offrant une sécurité absolue sur la correction du programme après la nouvelle déclaration.
Pour tester (C) il suffit en effet d'explorer les déclarations du bloc B(ND(K)) et de ses ancêtres.

R5 : Supposons que, pour détecter les conflits sémantiques, l'on veuille tester une condition moins restrictive que (C), telle que, par exemple :

(C) "Aucune occurrence d'un objet global n'est captée par la déclaration de v".

Le test de (C') nécessiterait l'exploration de la partie instruction du bloc B(ND(K)) chaque fois qu'un objet de même nom que v serait découvert dans les déclarations d'un bloc ancêtre.

Rappelons que, dans ce cas là, l'existence de l'instruction "with" en Pascal complique considérablement le problème.
Une condition telle que (C') pénaliserait donc l'utilisation de DECVAR, ce qui serait grave pour une commande utilisée systématiquement sous MENTOR.

I.3.2.2 - Contrôle des vérifications par l'utilisateur

Il arrive parfois que l'on ait envie de déclarer des variables locales à un bloc ayant le même nom que des objets globaux à ce bloc. C'est le cas, par exemple, pour le nom des compteurs des boucles "for", ou bien lorsque l'on réutilise une portion de code et qu'il faut déclarer localement les objets qui apparaissent dedans.

Nous avons vu dans la section 2 que l'on pouvait contrôler le positionnement des déclarations en passant un argument ND à DECVAR. On peut également contrôler les vérifications faites par DECVAR en lui passant un deuxième argument. Ce dernier indique à la commande DECVAR jusqu'à quel niveau elle doit remonter dans la chaîne des ancêtres du bloc B(ND(K)) pour effectuer les vérifications.

Nous appelerons cet argument "Niveau de Vérification" ou NV.

L'argument NV est une instruction MENTOL qui indique un chemin dans l'arbre du programme à partir de la position ND(K).

La position NV(ND(K)) indique à DECVAR le niveau à ne pas dépasser pour effectuer les vérifications. La condition C n'est alors testée que pour les objets déclarés à l'intérieur du bloc B(NV(ND(K))) c'est-à-dire du bloc le plus interne contenant la position NV(ND(K)). Si ce bloc ne contient pas le bloc B(ND(K)), DECVAR se ramène à l'option à défaut et signale une anomalie.

I.4 - Utilisation de DECVAR

Un appel de la commande DECVAR peut avoir trois formes :

.DECVAR	Pas d'arguments : options par défaut.
.DECVAR<ND>	Le niveau de déclaration est contrôlé et la vérification suit l'option par défaut.
.DECVAR<ND,NV>	Les niveaux de déclaration et de vérification sont contrôlés.

ND et NV sont des instructions du langage de commande de MENTOR et peuvent, en particulier, faire intervenir des commandes de positionnement.

I.4.1 - Exemples d'appels avec arguments

i) .DECVAR<.PROC,U*> : Equivalent à l'appel de DECVAR sans arguments.

En effet, la procédure PROC positionne le pointeur courant à la racine du bloc courant. La déclaration créée le sera donc dans ce bloc. Le deuxième argument, U*, désigne la racine du programme et les vérifications seront donc faites en remontant jusqu'à la racine.

Rappelons que la commande "u" de MENTOR est la primitive qui provoque la remontée du pointeur courant d'un niveau dans l'arbre. Toute commande MENTOR peut être suivie d'un facteur de répétition entier. Le facteur de répétition "*" indique de répéter la commande tant que c'est possible. La commande "U*" déplace donc le pointeur courant sur la racine de l'arbre.

ii) .DECVAR<.PROC,.PROC> : les vérifications ne sont faites que dans le bloc courant. Seuls les conflits syntaxiques seront donc détectés.

iii) .DECVAR<.FPROC> : La commande FPROC demande le nom d'un bloc et positionne le pointeur courant à la racine de ce bloc. Cet appel de DECVAR commence donc par demander un nom de procédure ou fonction et crée la déclaration dans celle-ci. Pour les vérifications l'option par défaut est appliquée.

iv) .DECVAR<.FPROC,.PROC> : Même chose que dans l'exemple précédent mais les vérifications sont faites uniquement à l'intérieur du bloc dans lequel les déclarations sont positionnées.

I.4.2 - Exemples de commandes "utilisateur" écrites à partir de DECVAR

Comme toutes les commandes MENTOR, la commande DECVAR peut-être utilisée directement où à l'intérieur de commandes programmées par l'utilisateur. Les formes d'appel des commandes sont strictement les mêmes dans les deux cas.

Supposons qu'un utilisateur désire changer les options par défaut. Le plus simple pour lui est de se créer sa propre commande de déclaration de variables qui appelera DECVAR avec les arguments qui lui conviennent.

Par exemple, un utilisateur qui considère que les vérifications par défaut sont un peu brutales peut définir une commande LOCALDECVAR de la façon suivante :

```
def  LOCALDECVAR,
   begin
     DECVAR(.PROC,.PROC)
   end ;
```

Ce qui, en MENTOL, s'écrit :

```
.def<.LOCALDECVAR, .DECVAR<.PROC,.PROC>>.
```

Remarquons que la commande LOCALDECVAR définie ci-dessus est plus efficace que la commande DECVAR utilisée avec les options par défaut. En effet, LOCALDECVAR n'effectue des vérifications que dans le bloc courant : son temps de réponse ne dépend donc que de la taille de ce bloc. A l'inverse, lorsque DECVAR est utilisée avec les options par défaut, le temps de réponse est lié à la taille de l'ensemble du programme manipulé.

Supposons maintenant que cet utilisateur ait très souvent besoin de déclarer des nouvelles variables globales à son programme. Il peut alors définir une commande qui, systématiquement, ira déclarer les variables au plus haut niveau :

```
def  GLOBALDECVAR,
   begin
     DECVAR(u*)
   end ;
```

Ce qui, en MENTOL, s'écrit :

```
.def<.GLOBALDECVAR, .DECVAR<u*>>.
```

I.5 - Pourquoi utiliser DECVAR ?

La commande DECVAR permet de déclarer des nouvelles variables sans qu'il soit nécessaire de se déplacer dans le programme à la recherche de l'endroit approprié pour faire ces déclarations. Le programmeur fait donc l'économie de plusieur "allers-retours" dans le programme entre les zones de déclarations et la partie du code sur laquelle il est en train de travailler. Il devient ainsi facile de déclarer les variables à l'instant même où elles s'avèrent utiles : le programmeur n'a plus à prévoir à l'avance les variables dont il aura besoin et il ne risque plus d'oublier de déclarer celles qui n'étaient pas prévues.

Lorsqu'il utilise DECVAR, le programmeur est également assuré que les nouvelles déclarations seront positionnées dans la zone de déclaration qui correspond à l'endroit du programme sur lequel il est en train de travailler. De plus, si le nom de certaines des nouvelles variables créent des conflits, le programmeur le sait immédiatement et peut donner d'autres noms à ces variables.

Des commandes analogues à DECVAR existent pour les déclarations de types (DECTYPE) et de constantes (DECCONS).

Un exemple de session MENTOR utilisant DECVAR et d'autres commandes est donné en appendice.

II - NORMALISATIONS, TRACES

Certaines commandes MENTOR font des calculs sur l'ensemble du programme. Pour les illustrer nous avons choisi de décrire ici deux commandes de normalisation et deux commandes de trace.

Une discussion complète sur les normalisations de programme et les traces possibles avec MENTOR peut être trouvée dans [MEL80]. Rappelons simplement que la philosophie des normalisations sous MENTOR est de mettre les programmes sous une forme plus simple et plus homogène et, par conséquent, plus lisible pour le programmeur et plus aisément manipulable par des outils programmés.

II.1 - Alpha-conversion

L'alpha-conversion est la transformation qui, à partir d'un programme P, crée un programme Q équivalent à P mais dans lequel deux objets distincts ont des noms distincts. Certains objets de P apparaissent donc dans Q sous un autre nom. Les nouveaux noms sont créés par le système en suffixant les anciens par un chiffre : ils sont donc très proches des noms originaux.

L'implémentation de l'alpha-conversion exige une étude très approfondie des règles de portée des identificateurs dans le langage traité. Pour le cas du langage Pascal cette transformation est définie formellement dans [MEL80] comme une interprétation non standard d'un programme [DON78a] en utilisant les principes de la sémantique dénotationnelle ([MIL76][GOR78][TEN76]) et une définition dénotationnelle de Pascal faite par Tennent [TEN78].

L'alpha-conversion simplifie considérablement la structure des programmes. En effet, sur un programme ainsi traité, chaque nom d'objet apparaît au plus une fois dans une déclaration. Les règles de liaison d'une occurrence d'un identificateur à une déclaration sont donc simplifiées.

La lisibilité des programmes s'en trouve en général améliorée. Tous les outils programmés qui utilisent la liaison entre un identificateur et sa déclaration sont beaucoup plus simples si on suppose que le programme à préalablement été transformé par alpha-conversion.

Notons que la commande d'alpha-conversion disponible sous MENTOR garde les informations nécessaires pour permettre de ramener le programme à sa forme initiale. La commande qui effectue cette transformation inverse est également disponible sous MENTOR. Ces informations sont attachées au programme alpha-converti sous la forme d'un commentaire structuré.

Ce commentaire permet aussi à la commande d'alpha-conversion de reconnaitre les programmes qui ont déjà été traités et donc de ne pas lancer inutilement des calculs qui sont tout de même assez coûteux.

A cette occasion, rappelons que, sous MENTOR, la notion de commentaires structurés ou "attributs de programme" est importante et largement utilisée. Il est en effet possible de coder des informations et de les attacher à un endroit du programme pour les réutiliser par la suite. Ces informations peuvent être visibles ou cachées. Dans le cas où elles sont visibles, elles apparaissent à la décompilation comme des commentaires. Il est parfois agréable et pratique de pouvoir cacher certaines informations, uniquement destinées a être réutilisées par des outils programmés, pour ne pas salir le code sur lequel on travaille.

II.2 - La suppression des déclarations inutiles

Au cours de la mise au point et de la maintenance d'un programme, celui-ci subit en général de nombreuses transformations. Il en résulte, au bout d'un certain temps une dégradation du code. En particulier dans le cas de gros programmes utilisés depuis longtemps et maintenus par plusieurs personnes, chacune d'entre elles hésitera à supprimer des objets existant dans ce programme. Il est alors fréquent que certains identificateurs déclarés, ou certaines étiquettes soient, en fait, inutiles.

La procédure de suppression recense tous les objets (variables, constantes, types, procédures, fonctions, étiquettes) déclarés dans le programme et qui ne sont jamais utilisés.

Leur destruction peut être effectuée automatiquement lors du recensement ou bien se faire par la suite au cours d'un dialogue, si l'on désire en conserver certains pour un usage ultérieur. Dans le cas où cette deuxième option est choisie, les entités inutiles sont présentées une par une à l'utilisateur qui indique s'il désire ou non les détruire. Avant d'effectuer la destruction, le système demande confirmation de la réponse s'il s'agit d'une procédure, d'une fonction ou d'un objet exporté. Dans tous les cas, ce qui a été détruit peut être récupéré tant que l'on n'est pas sorti de la session MENTOR en cours.

II.3 - Traces et profils d'exécutions

II.3.1 - Traces

Les outils de traces sont tous ceux qui rajoutent des instructions dans le programme pour obtenir à l'exécution des informations sur les endroits par lesquels le contrôle est passé.

Toutes les instructions rajoutées par les outils de trace disponibles sous MENTOR peuvent être ensuite éliminées automatiquement. Cela est réalisé très simplement en mettant un attribut à chaque instruction créée. Cet attribut permettra ensuite à la procédure de suppression de reconnaitre les instructions qui doivent être détruites.

La commande .TRACE demande le nom des procédures et fonctions à tracer. On peut également lui demander de tracer toutes les procédures et fonctions du programme. Elle ajoute au début du corps de ces sous-programmes une instruction d'écriture d'un message indiquant que l'on entre dans ce sous-programme. L'instruction rajoutée est :

```
writeln ('Entrée dans ',' nom de la procédure') ;
```

La commande de trace peut aussi être utilisée de façon à ce que l'utilisateur précise lui-même l'instruction à rajouter au début du corps des sous-programmes qu'il désire tracer.

II.3.2 - Profils

Les outils de profils d'exécution permettent d'obtenir des informations sur le nombre d'exécutions d'un point précis du programme. Un programmeur peut ainsi déterminer quels sont les fragments du programme qu'il est important d'améliorer.

La commande de profil .COUNT , exécute les opérations suivantes :

- demande un nom pour le compteur et vérifie que ce nom ne crée pas de conflits (Voir I.3.1). En cas de conflit, elle demande un autre nom.

- déclare ce compteur de type entier dans le bloc le plus externe.

- rajoute une instruction au début du programme pour initialiser le compteur à zéro.

- rajoute une instruction d'incrémentation du compteur devant l'instruction à compter qui est la position courante dans le programme. (Il y a échec si la position courante n'est pas une instruction).

- rajoute à la fin du programme une instruction d'écriture indiquant le nom du compteur et sa valeur.

Avec la commande .COUNT il est très simple de compter le nombre d'exécutions de toutes les instructions d'une certaine forme. Il suffit pour cela d'employer cette commande dans une boucle du langage MENTOL.

Exemple :

L'instruction MENTOL suivante introduit des instructions permettant de compter le nombre de fois que sont exécutées toutes les instructions while d'un programme PASCAL. Un compteur différent sera utilisé pour chacune d'entre elles :

.FORALL < @ while, (S2 ; .COUNT)>.

Remarque :

Si l'on désire compter le nombre d'exécutions de toutes les affectations apparaissant dans un programme, il faut raffiner un peu cette boucle MENTOL. En effet, la commande .COUNT rajoute elle-même des affectations dans le programme et il ne faut pas les prendre en compte.

II.3.3 - Suppression des instructions créées par les traces et profils

La commande .SUPP élimine toutes les instructions du programme qui ont été rajoutées par l'une des commandes de trace ou de profil. Pour permettre cette suppression, on doit trouver un moyen de distinguer rapidement et de façon sûre les instructions rajoutées de celles du programme. La simple adjonction d'un attribut à ces instructions permettrait de les distinguer de façon sûre mais par un processus très long : il serait en effet nécessaire d'examiner chaque instruction puis ses attributs éventuels. Pour que ce processus soit rapide, chaque instruction rajoutée par l'une des commandes de trace ou de profil est étiquetée, et l'étiquette reçoit sa propre valeur en attribut. Pour la suppression, il suffit alors de faire une recherche des instructions étiquetées, et de supprimer celles dont l'étiquette a sa propre valeur en attribut.

La commande .SUPP supprime également les listes d'instructions (c'est-à-dire les begin...end) qui ne sont plus nécessaires une fois les instructions de profil supprimées.

Les déclarations des étiquettes qui ont servi à marquer les instructions, et les déclarations des compteurs sont elles aussi supprimées.

Remarque :

Cette méthode d'utilisation d'un attribut pour décider si une instruction doit être détruite ou pas est suffisement sûre pour être utilisée sans problèmes. En effet, il faut noter les deux points suivants :

i) L'attribut, qui, à la décompilation, se présente comme un commentaire du programme, est sur un noeud (l'étiquette) auquel il n'a pu être attaché que volontairement par l'utilisateur : aucun commentaire n'est attaché à cet endroit par le processus d'analyse et de construction d'arbres.

ii) A la différence d'un commentaire classique, cet attribut n'est pas un morceau de texte. C'est un sous-arbre qui représente la constante entière correspondant à l'étiquette. Il n'y a donc pas de confusion possible entre cet attribut et un morceau de texte mis en commentaire du programme même dans le cas où ce dernier aurait la même représentation lorsque le programme est décompilé.

Il est donc possible de "tromper" la commande .SUPP en créant "à la main" un attribut qu'elle reconnaîtra, mais il faut l'avoir fait intentionnellement.

APPENDICE 1

LES COMMANDES DE HAUT NIVEAU EXISTANT DANS MENTOR

La plupart des commandes de haut niveau qui existent dans MENTOR pour manipuler des programmes Pascal sont listées ici. Une explication succinte est fournie pour chacune d'entre elles. Toutes ces commandes sont des procédures écrites en MENTOL. Pour une description du langage MENTOL lui-même, et des primitives du système MENTOR (telles que les primitives de manipulation de fichiers, de paramétrisation du système etc...) se reporter à [HUE77] ou [MUL80].

Rappelons que sous MENTOR, le nom des commandes commence toujours par le caractère point ('.'). Par conséquent, le nom réel de chacune des commandes listées ci-après est obtenue en préfixant le nom indiqué par un point. Lorsqu'une commande prend des arguments, ceux-ci, sont mis entre les symboles '<' et '>'.

Les arguments de ces commandes sont de deux types :

- variables MENTOL désignant un sous-arbre

  (exemple : @P, @Q)

- Instructions MENTOL désignant des actions à exécuter

  (exemple : .ACTION)

Les arguments indiqués entre accolades sont facultatifs.

BODY : Déplace le pointeur courant sur le corps de l'instruction ou du bloc dans lequel se trouve la position courante.

CALL : Demande le nom de deux sous-programmes et indique si le premier peut appeler le deuxième. Si il existe, le chemin trouvé entre ces deux sous-programmes dans le graphe des appels est gardé dans la variable MENTOR '@path'.

CASE : Si la position courante est à l'intérieur d'une instruction 'case', indique dans quel cas du case elle se trouve.

CLEAN : Nettoyage du programme : suppression des instructions vides et des 'begin...end' inutiles.

CLOSURE : Calcul de la fermeture transitive du graphe des appels. Le graphe est construit si il ne l'était pas déjà.

COMBODS : Rajoute un commentaire au début du corps des sous-programmes dans lesquels il y a des déclarations de blocs internes.

COMPILE : Sauvegarde du programme manipulé sous forme d'arbre et sous forme de texte avec mise à jour de la version, puis compilation par le compilateur disponible sur l'installation. Au retour de la compilation l'utilisateur est remis sous MENTOR dans le même état que lorsqu'il a appelé la procédure de compilation. Actuellement cette procédure n'existe que dans la version de MENTOR qui se trouve sur MULTICS.

COMPROCS : Rajoute, à la fin des procédures et des fonctions, un commentaire rappelant leur nom.

CON : Remonte sur la zone de déclaration de constantes du bloc courant, si elle existe. Echoue si cette zone n'existe pas.

COND : Va sur la condition d'un WHILE, IF, REPEAT, FOR. Echoue si la position courante n'est pas l'une de ces instructions.

COUNT : Cette procédure introduit dans le programme les instructions et les déclarations nécessaires pour compter le nombre de fois qu'un certain point du programme est exécuté. Echoue, si la position courante n'est pas une instruction. La procédure .COUNT demande un nom de compteur à l'utilisateur, déclare ce compteur au plus haut niveau avec le type entier, crée une instruction d'initialisation à zéro en début de programme, et crée une instruction d'incrémentation de ce compteur devant le point de programme à compter. Une instruction pour écrire la valeur finale du compteur est introduite à la fin du programme. Tout ce qui a été créé par .COUNT peut être détruit automatiquement par la procédure .SUPP.

CROSSREC : Indique si deux procédures P et Q appartiennent à un même cycle du graphe des appels (détection des récursions croisées).

CTX : Remonte jusqu'à un for, while, repeat, case, begin...end ou bloc. Echoue si ne trouve pas l'une de ces constructions.

DECCONS {<.ND,.NV>} :

Déclaration d'une nouvelle constante. Cette procédure demande le nom de la constante à déclarer, puis sa valeur. La zone de déclaration des constantes est créée si elle n'existait pas. Les arguments ND (niveau de déclaration) et NV (niveau de vérification) peuvent être utilisés pour contrôler le positionnement et les vérifications faites lors de cette déclaration.

DECFORWARD {<.ND>} :

Création d'une déclaration en avant pour une procédure ou une fonction. Il est vérifié qu'une telle déclaration n'existait pas déjà pour cette procédure ou cette fonction. Les arguments sont mis en commentaire au niveau de la déclaration complète.

DECTYPE {<.ND,.NV>} :

Déclaration d'un nouveau type. Cette procédure demande le nom du type à déclarer et sa valeur. La zone de déclaration des types est créée si elle n'existait pas déjà. Les arguments ND et NV peuvent être utilisés pour contrôler le positionnement et les vérifications faites lors de cette déclaration.

DECVAR {<.ND,.NV>} :

Déclaration d'une liste de variables. Cette procédure demande une liste de variables et une expression de type. La zone de déclaration de variables est créée si elle n'existait pas déjà. Les arguments ND et NV peuvent être utilisés pour contrôler le positionnement des déclarations et les vérifications faites.

DEL : Interface de suppression des objets inutiles détectés par la procédure de recencement des objets inutiles : CHECK-UNUSED.

DETAILS : Documentation interactive sur les procédures disponibles en MENTOR. Cette procédure demande le nom d'une procédure MENTOL et fourni une explication sur son fonctionnement. Disponible uniquement sur MULTICS pour l'instant.

DIRECTCALL : Détermine les sous-programmes appelés par un sous-programme donné.

DVAR : Transforme un paramètre par référence en paramètre par valeur dans un en-tête de procédure ou de fonction. La position courante doit être une déclaration de paramètre.

EXIT : Sortie de la session en cours de MENTOR avec sauvegarde, sous forme d'arbre et sous forme de texte, du programme manipulé lors de cette session. Création (ou mise à jour) d'un commentaire en tête de ce programme indiquant le nom du programmeur et la date de la dernière modification.

FPROC {<.NC>} :

Demande le nom d'une procédure ou fonction et cherche celle-ci dans la suite du programme. Echoue si il n'y a pas de procédure ou fonction de ce nom. L'argument NC peut être utilisé pour indiquer à quel endroit la recherche doit commencer. Cette facilité est très utile lorsque qu'il existe plusieurs procédures ou fonctions ayant le même nom. L'argument NC doit être une instruction de déplacement à partir de la position actuelle du pointeur courant. En cas d'échec, la position courante est restituée à l'endroit où elle était avant l'appel. Indique si le sous-programme cherché a une déclaration en avant (FORWARD).

GINFO : Consultation interactive du manuel MENTOR. Disponible uniquement sur MULTICS pour l'instant.

GRAPH : Construction du graphe des appels des sous-programmes définis par l'utilisateur. Pour chaque sous-programme SP, une ligne de la forme SP(SP1,SP2,...,SPn) est écrite. Une telle ligne indique que le sous-programme SP appelle les sous-programmes SP1,SP2,...,SPn. Le graphe est ensuite conservé durant toute la session et peut donc être réutilisé.

GRAPHCOMP : Même principe que pour la procédure GRAPH, mais construction du graphe des appels pour TOUTES les procédures et fonctions utilisées dans le programme (y compris les procédures prédéfinies telles que WRITE, READ, etc...).

HELP : Appel de l'assistance interne à MENTOR. Des informations sur l'état du système sont disponibles. Ces informations sont accessibles par un système de mots clefs.

INVERSE : Opération inverse de l'alpha-conversion. Le programme doit avoir été transformé par alpha-conversion auparavant. Il est alors ramené à son état antérieur en utilisant les informations construites et attachées au programme, sous forme d'attributs, par la procédure d'alpha-conversion : RENAME.

JUMPOUT : Détecte toutes les instructions de saut (goto) qui provoquent la sortie d'un ou de plusieurs sous-programmes. C'est le cas chaque fois qu'une instruction de saut fait référence à une étiquette qui ne se trouve pas dans le même sous-programme que l'instruction de saut. Les sauts externes sont interdits par de nombreux compilateurs Pascal.

LAB : Remonte sur la zone de déclaration d'étiquettes du bloc courant. Echoue si cette zone n'existe pas. En cas d'échec la position courante est restituée à l'endroit où elle se trouvait avant l'appel.

LABEL : Etiquetage d'une instruction et déclaration de l'étiquette. Cette procédure demande un nom pour l'étiquette, vérifie qu'elle n'existe pas déjà et déclare cette étiquette en créant la zone de déclaration des étiquettes si c'est nécessaire.

NORMALIZE : Normalisation des programmes. Cette procédure appelle successivement les procédures CLEAN, COMBODS, et COMPROCS.

PROC : Remonte à la procédure ou fonction englobante.

PROFILE : Permet de faire des statistiques sur l'exécution d'un programme.

PUTDEF : Déclare un identificateur en 'def' (exporté). La position courante doit être une déclaration de variable.

PUTREF : Déclare un identificateur en 'ref' (importé). La position courante doit être une déclaration de variable.

PUTVAR : Transforme un paramètre par valeur en paramètre par référence dans l'en-tête d'un sous-programme. La position courante doit être une déclaration de paramètre.

RENAME : Alpha-conversion d'un programme. L'alpha-conversion est la procédure qui transforme un programme P en un programme Q, équivalent à P, mais dans lequel deux objets distincts ont des noms distincts. L'alpha-conversion garde les informations qui seront nécessaires si l'on désire, par la suite, ramener le programme à sa forme initiale (voir la procédure INVERSE). Ces informations sont accrochées au programme sous la forme d'un attribut qui contient la liste des substitutions à faire pour revenir à l'état initial du programme. La procédure INVERSE n'a donc plus qu'à appliquer ces substitutions.

REPLACEID : Cette procédure permet de remplacer certaines occurrences d'un identificateur A par un identificateur N. Elle demande le nom de l'ancien identificateur A, puis le nom du nouveau N. Ensuite, pour chaque occurrence de A, dans le sous-arbre courant, la main est passée à l'utilisateur qui a alors le choix entre trois réponses :

- $: pour remplacer cette occurrence de A par N et continuer,
- $- : pour ne pas modifier cette occurrence de A et continuer,
- $-3 : termine la commande REPLACEID sans modifier l'occurrence courante de A.

SAT : Suppression de certains types anonymes (ceux qui introduisent des occurrences définissantes). Cette procédure peut demander les nouveaux noms à l'utilisateur ou créer ces noms à partir des composants de l'expression de type à nommer.

STRACE : Trace standard des procédures et des fonctions. Cette procédure demande les noms des sous-programmes à tracer et rajoute au début de leur corps une instruction d'écriture du nom du sous-programme. Possibilité de tracer tous les sous-programmes d'un seul coup.

SUPP : Suppression des instructions qui ont été ajoutées dans le programme par les commandes de traces et de comptes d'exécutions.

TRACE : Trace paramétrée des procédures et des fonctions. Cette procédure demande à l'utilisateur l'instruction qu'il désire rajouter au début des corps des sous-programmes.

TYP : Remonte sur la zone de déclaration de types du bloc courant. Echoue si cette zone n'existe pas. Ne bouge pas en cas d'échec.

CHECK-UNUSED : Détecte les objets déclarés et qui ne sont jamais utilisés. La liste des objets inutiles est donnée en sortie en précisant dans quel sous-programme se trouve chacun d'entre eux. Cette procédure suppose que le programme traité est correct en ce sens qu'il peut être compilé sans erreur : la procédure CHECK-UNUSED ne recherche les utilisations éventuelles des objets que dans les endroits ou ceux-ci peuvent être légalement utilisés. De plus, CHECK-UNUSED ne fait aucune vérification de types et ne détecte pas les objets non déclarés. La procédure CHECK-UNUSED fait beaucoup de calculs sur le programme et est par conséquent assez coûteuse.

VAL : Va sur la zone 'value' du module. Echoue si cette zone n'existe pas. Ne bouge pas en cas d'échec.

VAR : Remonte sur la zone de déclaration de variables du bloc courant. Echoue si cette zone n'existe pas. Ne bouge pas en cas d'échec.

WHOCALLS : Demande le nom d'un sous-programme et indique en sortie tous les sous-programmes qui appellent celui-ci. Cette procédure construit le graphe complet des appels (par GRAPHCOMP) si celui-ci ne l'était pas déjà.

WRAP : Transforme une instruction simple en une liste d'instructions à un élément, et rend la position courante sur cet élément.

WRITE : Sauvegarde du programme courant sous forme d'arbre et sous forme de texte avec mise à jour de la version.

Procédures de manipulations d'arbres indépendantes du langage

APL<.ACTION> : Boucle sur les fils du noeud courant. L'action .ACTION est appliquée à tous les fils du noeud courant.

.DEF<.NOM,.DEFINITION> :

Définition d'une nouvelle procédure MENTOL. Le premier argument .NOM est le nom de la nouvelle procédure définie. Le deuxième argument .DEFINITION est une expression MENTOL qui constitue le corps de la procédure .NOM.

FORALL<@PAT,.ACTION> :

Boucle sur tout l'arbre à partir de la position courante. L'action .ACTION est appliquée à tous les sous-arbres dont la structure est compatible avec celle du schéma pointé par la variable MENTOR '@PAT'. En cas d'échec de l'action sur l'une des occurrences du schéma '@PAT', FORALL s'arrête en échec et rend la main sur le sous-arbre qui a provoqué l'échec. En cas de succès, c'est-à-dire, lorsque l'action .ACTION réussit sur toutes les occurrences de '@PAT', la position courante est restituée à l'endroit où elle se trouvait avant l'appel.

FOREACH<@PAT,.ACTION> :

Boucle sur le sous-arbre courant. Même principe que FORALL mais sans sortir du sous-arbre pointé par le pointeur courant.

SORT<.KEY, @L> : Tri d'une liste suivant une clef. La liste @L est réordonnée suivant le critère indiqué par la clef '.KEY'.

UP<@OP> : Prend un opérateur en argument et remonte jusqu'à un opérateur de ce type. Echoue si cet opérateur n'apparaît pas parmi les ancêtres du noeud courant.

APPENDICE 2

Exemple de session	Commentaires
?	
?p*	Demande d'impression du programme en entier
program EXEMPLE;	
function INC(X:INTEGER):INTEGER;	
begin	
INC:=X+1	
end;	
function DEC(X:INTEGER):INTEGER;	
begin	
DEC:=X-1	
end;	
begin	
for I:=1 to MAX do	
begin	
READ(Y[I]);	
Z[I]:=INC(Y[I])+DEC(Y[I])	
end	
end.	
?.body;p*	Aller sur le corps de ce programme et l'imprimer en entier
begin	
for I:=1 to MAX do	
begin	
READ(Y[I]);	
Z[I]:=INC(Y[I])+DEC(Y[I])	
end	
end	
?.deccons	Declarer une constante dans le bloc courant
constant name:[IDENT]:max;	
[CST]:10;	
?.decvar	Declarer des variables dans le bloc courant
list of variables:[LVARBL]:y,z;	
[TYP]:array[1..max] of integer;	
?.decvar	
list of variables:[LVARBL]:y;	
Y ALREADY EXISTS, give another name:[VARBL]:i;	
[TYP]:1..max;	
?.var;p*	Aller sur la zone de declaration de variables, et l'imprimer.
var Y,Z:array[1..MAX]of INTEGER;	
I:1..MAX	
?	
?.strace	Demande de traces standard
Give the block name, all for all, stop to stop:	
[IDENT]:dec;	
[IDENT]:inc;	
[IDENT]:stop;	
?	Boucle Mentol: pour toutes les boucles FOR, compter le nombre d'executions de la premiere instruction du corps.
?.forall<@for,(.body;s1;.count)>	
Please, give me the counter name:[IDENT]:cptfor1;	
?u*;p*	

```
program EXEMPLE;
     label
          (*3*) 3,(*4*) 4,(*5*) 5;
     const
          MAX     =10;
     var  Y,Z:array[1..MAX]of INTEGER;
          I:1..MAX;
          CPTFOR1:INTEGER;

     function INC(X:INTEGER):INTEGER;
          label
               (*2*) 2;
          begin
          (*2*) 2: WRITELN('entering ','INC');
          INC:=X+1
          end;

     function DEC(X:INTEGER):INTEGER;
          label
               (*1*) 1;
          begin
          (*1*) 1: WRITELN('entering ','DEC');
          DEC:=X-1
          end;

     begin
     (*4*) 4: CPTFOR1:=0;
     for I:=1 to MAX do
          begin
          (*3*) 3: CPTFOR1:=CPTFOR1+1;
          READ(Y[I]);
          Z[I]:=INC(Y[I])+DEC(Y[I])
          end;
     (*5*) 5: WRITELN('CPTFOR1','=',CPTFOR1)
     end.
```

Programme modifie

```
?.write
EXEMPLE.polish  FILE CREATED
EXEMPLE.pascal  FILE CREATED
?p3
```

Sauvegarde et creation d'un commentaire prefixe standard.

```
(********************)
(*.LAST UPDATE :   .*)
(*04/28/81  23:56   *)
(*.BY :            .*)
(*Bertrand*)
(********************)
(**)
program EXEMPLE;
     ...;...;...;#;#;
     begin
     #;#;#
     end.
```

Nouvelle forme du programme (au niveau 3)

```
?.supp                                          Demande de suppression
?p*                                             des instructions de trace
                                                et de comptage.
(*********************)
(*.LAST UPDATE :    .*)
(*04/28/81  23:56    *)
(*.BY :             .*)
(*Bertrand*)
(*********************)
(**)
program EXEMPLE;
     const
          MAX     =10;
     var  Y,Z:array[1..MAX]of INTEGER;
          I:1..MAX;

     function INC(X:INTEGER):INTEGER;
          begin
          INC:=X+1
          end;

     function DEC(X:INTEGER):INTEGER;
          begin
          DEC:=X-1
          end;

     begin
     for I:=1 to MAX do
          begin
          READ(Y[I]);
          Z[I]:=INC(Y[I])+DEC(Y[I])
          end
     end.
?.fproc;p*                                      Recherche d'un sous
procedure or function name:[IDENT]:dec;         programme et impression.

function DEC(X:INTEGER):INTEGER;
     begin
     DEC:=X-1
     end
?.body;s1;p                                     Aller sur la premiere
DEC:=X-1                                        instruction du corps, puis
?s2 s2 c &                                      sur le 2ieme fils du 2ieme
[EXP]:2;                                        fils, et le changer par ce
?p                                          qui va etre entre au terminal
DEC:=X-2
?.proc;p*                                       Remonter au bloc courant

function DEC(X:INTEGER):INTEGER;
     begin
     DEC:=X-2
     end
?.exit                                          Sauvegarde avec maintient
EXEMPLE.polish  FILE OVERWRITTEN                de la version et sortie
EXEMPLE.pascal  FILE OVERWRITTEN                de Mentor.
EXIT LEV.1-PASCAL
```

REFERENCES BIBLIOGRAPHIQUES

ARS 79 J. Arsac,
"Syntactic Source to Source Transforms",
CACM, 22, 1, p. 43-54 (1979).

DON 75 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J.J. Levy,
"A structure oriented program editor : a first step toward computer assisted programming",
International Computing Symposium, North Holland Publishing Co (1975).

DON 78 V. Donzeau-Gouge,
"Utilisation de la sémantique dénotationnelle pour l'étude d'interprétations non standard",
3ième Colloque International sur la Programmation, Dunod (1978).

DON 79 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang,
"Introduction au système MENTOR et à ses applications"
Journées francophones sur la certification du logiciel, Genève (Janvier 1979).

DON 80 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang,
"Programming environments based on structured editors : the Mentor experience",
INRIA, Rapport de Recherche n° 26 (Juillet 1980).

GOR 78 M. Gordon,
"Notes on the descriptives techniques of Denotationnal Semantics",
(February 1978).

HUE 77 G. Huet, G. Kahn, P. Maurice,
"Environnement de programmation Pascal : Manuel d'utilisation sous Siris 7/8",
(Novembre 1977).

JEN 74 K. Jensen, N. Wirth,
"Pascal User Manual and Report",
Lecture notes in Computer Science 18, Springer Verlag (1974).

MEL 80 B. Melèse,
"Manipulation de programmes Pascal au niveau des concepts du langage",
Thèse de 3ième cycle, Université Paris 11 Orsay (1980).

MIL 76 R. Milne, C. Strachey,
"A theory of programming language semantics",
Chapman and Hall (1976).

MOR 79 E. Morcos-Oury,
"Etude des effets de bord des appels de procedures et de fonctions dans le langage Pascal",
Thèse de 3ième Cycle, Université Paris 11 Osay (1979).

MUL 80 "The Mentor Program Manipulation System",
Disponible dans la documentation du système MULTICS de l'INRIA.

TEN 76 R.D. Tennent,
"The denotational sematics of programming languages",
CACM August, Vol 19 n° 8 (1976).

TEN 78 R.D. Tennent,
"A denotational definition of the programming language Pascal",
(April 1978).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique